FPGA Photobooth Final Report

1st Diego Rodriguez
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
diegorod@mit.edu

2nd Ellie Rabenold
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
rabenold@mit.edu

3rd Kojo Anane-Fordjour
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
kananefo@mit.edu

***Abstract*—This document outlines a design for a photobooth system developed on a Nexys 4 DDR FPGA as a final project for 6.2050 at the Massachusetts Institute of Technology in Fall 2022. The FPGA Photobooth project consists of three key elements — image processing, a user interface, and a pen plotter. This report will detail the considerations and challenges that influenced the digital design of each facet of the system.**

I. Overview

This project mimics a photo booth that enables the user to take a photo, pass it through a filter of their choosing, and print it onto paper via the 2-axis pen plotter. Once the user has triggered a photo to be taken, they are presented with 6 options for to filter the grayscaled image; among the offered filters are dithering, a wave filter, and edge detection. The user manipulates the on-board buttons to make their selection with the VGA-based user interface. Then, the user follows a similar process to select the threshold value that is applied to the image as it is converted from grayscale to black and white. Once the picture has been processed, it is compressed and sent to the pen plotter, which draws it out onto a canvas.

II. Image Processing and Manipulation (Diego)

A. Filters

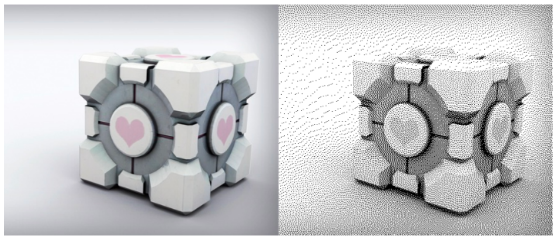


Fig. 1. The grayscale-dither filter applied to an image

- **Dither**: applies a Floyd-Steinberg Formula that converts the image to black and white using a pattern that dissipates pixels and emulates a grayscale picture while minimizing error by dispersing it across the picture. This is done by finding a pixel's "lowest error" pixel, and then adding a fraction of said error to the pixel to the right, as well as the 3 pixels under it. This was implemented using a single BRAM that continuously reads the offset for the pixel in front of the current pixel, and writes the total offset of the pixels for the next line, directly behind the current pixel, pipelined accordingly. We hoped that since this converts pixels to 1 bit, we could use it to save space on BRAMs, though we then learned afterwards that the minimum width is 8 bits.
- **Wave**: Applies a polynomial to the pixel's y-coordinates that moves the x-coordinates left or right, and also checks their updated position, and wraps them around back to the other side of the image as necessary to preserve dimensions and pixels. The polynomial uses signed variables that detect the y-coordinate location (which turned out to be hcount_sync) relative to the midpoint and bottom of the image, and moves the x-coordinate of the current pixel, checks if it's in bounds, and wraps around if necessary. The fact that hcount and vcount are irregular means that careful use of its BRAM is required to ensure the image is stored and read correctly, which can be confusing with the applied transformations on the image.
- **Ridge**: This one makes use of convolution, buffers, and kernels, adapted to work with grayscale pixels. It uses the same ridge kernel as lab 4b, outputting pixels as their value minus the value of the pixels surrounding them, meaning only pixels with a sufficiently high value compared to those surrounding it are seen.
- **Identity**: This one also uses the same approach, though it instead uses the blur convolution to appear less noisy.
- **Wave_y**: This works similar to the other wave filter, though it applies the polynomial to the x-coordinates and adds an offset to the y-coordinates. Once again, rotation and mirroring makes this tricky to make work correctly. As well, ensuring the multiplication can be done within a single clock cycle, and the resulting offset isn't so large as to completely disrupt the image is also required.

- **Negative**: since we started running out of space for BRAMs, this one reads from the exact same BRAM as identity, but instead outputs its inversion, allowing another filter to be used without consuming another BRAM.

B. Threshold

Based on the input, a threshold is chosen. If a pixel's value is above the threshold, it is evaluated as white, otherwise black. This converts to a 1-bit data size. The pixel data that is fed through the threshold is 7 bits and four options for thresholding are offered, each being roughly equidistant points between 0 and 127, the values we used for grayscale intensity.

```
case (thresh_mux)
    2'b00: pixel_out = (pixel_in > 25);
    2'b01: pixel_out = (pixel_in > 51);
    2'b10: pixel_out = (pixel_in > 77);
    2'b11: pixel_out = (pixel_in > 102);
endcase
```

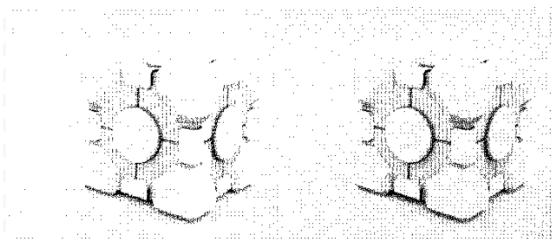


Fig. 2. Dithered image thresholded at 25 and 51

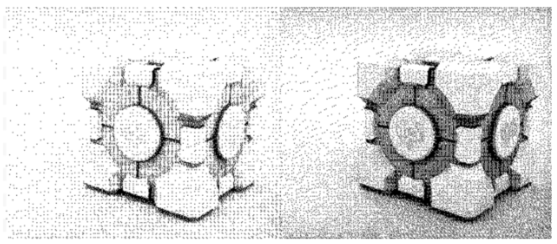


Fig. 3. Dithered image thresholded at 77 and 102

C. Average and Compression

In order to make an input small enough for the plotter to write on a small paper, we learned that we had to convert our images to 240x320 to 80x106. In order to do this while preserving the image, we created a convectional image compressor that averages 3x3 pixel values to a single pixel. Since at the point that the image is received by the plotter side, the pixels are 1 bit in size, the approach simply involves using a buffer to obtain a 3x3 kernel of pixels, and checking if it has 5 or more pixels with a value of 1. Then, we keep track of the hcount and vcount to only receive and store the pixels from every third hcount of every third vcount. This allows the plotter to use a smaller aspect ratio while still maintaining the general shape of the image.

III. User Interface (Kojo)

The user interface internally is a finite state machine that takes on a different display depending on the state of the program. Image sprites were utilized to show instructions on how to interact with the program. The camera module was also used to output a gray scale image onto the screen. The code for the UI is housed within a module named screen, which is divided into four main states: START, FILTERS, THRESHOLD, and SEND.

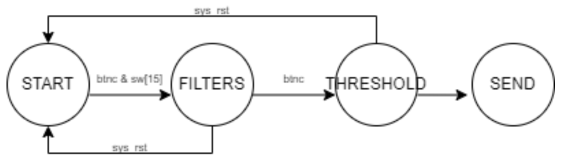


Fig. 4. The state machine for the user interface

- **START**: This is the initial screen a user will see. There is a 240x320 pixel gray scale live feed from the camera and also instructions on how to use the FPGA to operate the system. When sw[15] turns high, the live video becomes a still image. To proceed, the user needs to press btnc. When btcn goes high, the state is changed to FILTERS.
- **FILTERS**: In this state, some basic mathematical calculations are used to create 6 boxes and an arrow that is symmetric. Since the VGA monitor is 1024 pixels wide and each image is 240 pixels wide, on a y-axis, the vertical columns for the images are between [50, 290), [390, 630), and [730, 970). The monitor is 768 pixels wide and each image is 320 pixels. To also allow a 100x100 pixel arrow image sprite to run across the middle of the screen, the x-axis values for the horizontal rows for the images are between [26, 346) and [446, 776). This setup leaves 26 pixels vertically at the very top of the screen which is enough to display the instructions for this filters state. The arrow image sprites are moved around based on input from btnl and btnr, which is how the user decides which filtered image they want to process. Once the desired filter is picked, the user can advance by pressing btnc.
- **THRESHOLD**: In this state, the user is shown their selected filtered image at four different levels of thresholding. To achieve this, filter selection is tracked at the previous state and that filtered image is thresholded and displayed. Again, by interfacing with btnl and btnr, the user is able to select which thresholded image they want to continue with. Once satisfied, btnc advances the user to the next state.
- **SEND**: This state serves as an end state. Here, the filtered image and the thresholded black and white image for that filter is saved and ready to connect with the rest of the system such that the plotter can begin drawing the image.

The state machine within screen served to be the boiler plate that would hold the filtered images for user selection.

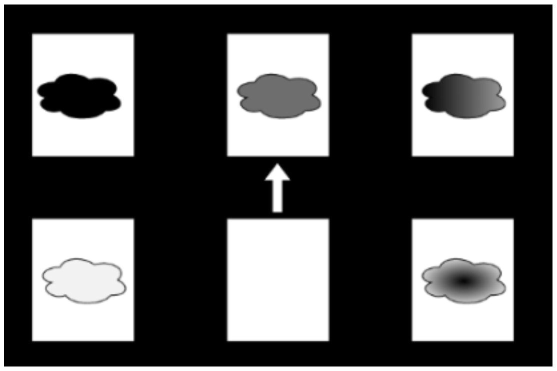


Fig. 5. Simplified example of filter select screen

Within the top_level module were the BRAMs which held the filtered data. Additionally, was the vga module which is what allowed pixels to be put on the screen. By using the same vga instantiation for both the user interface and filtered images, it was possible to display both instances at the same time. Calculations were performed to get the specific addresses for filters in specific boxes. Since the VGA screen is 1024 pixels by 768 and each image was 240 pixels by 320 pixels and there are six different options, it was determined that box 1 should occupy a hcount of [50, 240) and a vcount of [26, 346). Since the data is stored in a BRAM, we can read the address by shifting from the BRAM by shifting the current hcount and vcount by the image start offset. However, due to a design choice in an earlier part of the system, the camera's pixel data is placed into the BRAM in a reverse order. More generally, if the desired location for a box is known, addresses can be mapped by this general equation:

$$read\_addr = (img\_width * img\_height - 1) - \\ ((hcount - start\_h) * img\_width + (vcount - start\_v)) \quad (1)$$

IV. Plotting the Picture (Ellie)

A. Hardware

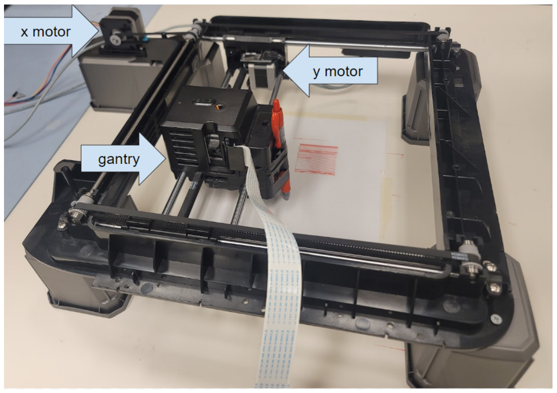


Fig. 6. The 2-axis pen plotter

The 2-axis pen plotter system consists of an X-Y frame borrowed from a Flashforge Inventor II 3D printer. A ballpoint pen of respectable quality (ideally a Pilot G2 07) is mounted with tape to the front end of the gantry block. A combination of linear rods, linear bearings, pulleys, and timing belts enable the gantry to traverse the coordinate plane. The gantry is actuated by two bipolar stepper motors mounted to the plotter's frame, and each stepper motor is driven by a L298N motor driver. The motor driver modules are powered by 12V and are connected to the JC and JD PMOD pins on the Nexys 4 DDR board.

B. Motor Control

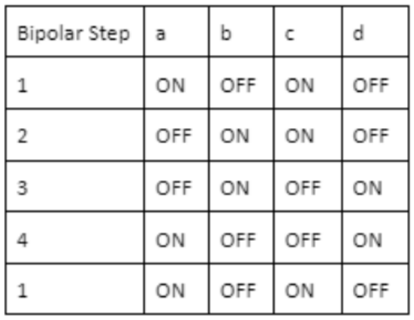
The plotter hardware informed a significant portion of the digital design that commands the plotter. The most salient place this occurred is in how the stepper motors are actuated.

- **Pulse-width Modulation:** The plotter motors are controlled by L298N motor drivers that accept PWM input signals to affect the stepper motors' speed and direction. The top level module initializes with a 100MHz clock, but to prevent clock domain crossing with the camera clock, the plotter module operates at 65MHz. An internal counter in the PWM generator module acts as a clock divider to generate a 50% duty cycle signal at the desired frequency. Because the plotter does not need to vary its speed, a set 50% duty cycle is sufficient. Although a frequency as low as 10Hz is sufficient to operate the plotter, at that speed a drawing would take at minimum two hours to complete. Through experimental testing a 40Hz PWM signal was found to be appropriate.
- **Phase Sequencing:** Each phase of the stepper motor must be energized in a specific sequence to generate a step. To spin the motor clockwise, the phases are pulses by moving forward through the sequence. By stepping backwards through the sequence, the motor can be reversed.

  The plotter control module uses one state machine per motor. A global counter variable keeps track of the step in the sequence. By incrmenting or decrementing each counter variable, the state machine is able to step the plotter in either direction. One feature of the plotter is that it can be paused mid-drawing. By tracking the sequence steps, the plotter can resume drawing without issue.

  To operate at the desired frequency, the motor pulse state machines only write to the PMOD pins on the rising edge of the 40Hz PWM signal.
- **Scaling the Drawing:** Another decision driven by the hardware is how the image should be scaled on the canvas. The design was bounded by two constraints - the dimensions of the plotter's 'build volume,' and by the distance that a single stepper pulse moves the pen. The gantry can traverse at most a 5.5" x 5.5" square on the canvas. Determined experimentally, this translated to roughly 1000 steps of of the pulse sequence. From this, a 3x scaling of the image's 320x240 dimensions seemed appropriate.

  However, it turns out that even with 3x scaling, three plotter steps (representing an edge of a pixel), is not



Bipolar Step	a	b	c	d
1	ON	OFF	ON	OFF
2	OFF	ON	ON	OFF
3	OFF	ON	OFF	ON
4	ON	OFF	OFF	ON
1	ON	OFF	ON	OFF

Fig. 7. The pulsing sequence to drive a motor

enough to visually separate rows. Through trial and error, it was determined that nine plotter steps are sufficient to distinguish between different rows of pixels. But, 240 pixels per row at nine steps per pixel is outside the plotter's dimensions. To solve this, the final black and white image is put into a 9x9 pixel convolution module that determines whether the region is mostly black or mostly white. Here, the image is compressed by 3x - its new dimensions are 106x80. This compressed image is written into a BRAM, then its data is sent to the plotter pixel by pixel.

- **Representing a Pixel:** One consideration when designing the plotter system was how to represent a pixel versus an empty space. Given that the pen stays on the canvas the whole time, an empty pixel necessarily must be represented by a straight line of 9 steps of the stepper motor sequence. A filled pixel is represented with an 'X' shape by pulsing both steppers in a specific sequence. See the image below for a visual representation of this.

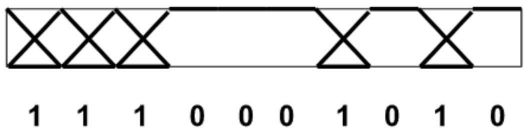


Fig. 8. Example sequence

V. Plotter FSM Design

The plotter design is governed by a state machine that is responsible for controlling the counters used in the stepper motor sequence. For the plotter to operate without manual input, it has to be capable of four key functions.

- It needs to be able to draw a filled pixel.
- It needs to be able to draw an empty pixel.
- It needs to be able to shift the carriage from the right end of the canvas back to the left end of the canvas.
- It needs to be able to shift the carriage down to the next row.

These plotter functions are encapsulated within a state machine that is engaged when the processed image is ready to be drawn. The initial state depends on the value of the image's first pixel. From there, the image's pixels are output onto the canvas while a horizontal position tracker monitors whether the plotter is approaching the edge of the canvas. If this triggers, the system enters the CARRIAGE RESET state, which pauses the receipt of new data as the carriage is pulled back across the horizontal axis.

Once the horizontal counter has reached 0, meaning the carriage is back at the left edge of the canvas, the state machine then enters the SHIFT LINE state, which moves the carriage down to the next row. Then, the state machine enables pixel data receipt, and moves to the state that corresponds to the received pixel.

A vertical position tracker monitors the position of the carriage as it proceeds down the canvas. Once the carriage has reached the bottom right corner of the canvas - the last column in the last row, the vertical counter triggers the DRAWING DONE state. At this point, the user can cut the power supply, remove the plotter, and take their finished drawing.

VI. Evaluation

A. Memory

Due to the nature of the project, it was necessary to store lots of image data. For this, the FPGA's block RAM was utilized. The process initially stores the camera's pixels to a BRAM, which is 320x240 pixels each 12 bits wide. From there, five other BRAMs are utilized to store the various filters, each one with a height of 320*240 and a depth of 7 pixels. In total, this is close to 3.6 million bits of data stored on the BRAM at a given time. Additionally, a few image sprites are used within the project, but this storage is almost negligble compared to the pixel data. According to specifications of the FPGA, the BRAMs can store close to 4.8 million bits, meaning that almost all of the storage was used. In fact, while testing later builds, the FPGA did run out of memory, so there may be extra storage that is unknown. This is not sustainable for future extensions since there is a risk of running out of space. One possible fix for smalller variables might be to hold multiple variables in the same address on a BRAM, since they each use a minimum of 8 bits in width. As well, with some extremely careful pipelining, it may have been possible to use some filters without storing their outputs in a BRAM.

B. Timing

By analyzing the build logs, it was discovered that the Post Router Timing for the project had a worst negative slack (WNS) of -0.076 and a total negative slack (TNS) of -0.118. During Physical Synthesis Initialization, there was a WNS of -2.03 and TNS of -329. This indicates that the signal path is slower than the required time for most stages of the build. Although timing constraints are not met, the functionality of the design was not affected heavily. Apart from button pressing and switching, there is not other places of the code where it is mandatory that timing is sufficient.

C. Use Cases

The initial goal was not met, however it was very close. Individual components were for the most part sound, but integrating all the pieces took quite longer than expected. A stretch goal included using some sort of clustering to better decide how to draw pixels. This would have been a great improvement because currently even with the plotter scaling a 240x320 image down to a 80x106, it takes around 30 minutes to draw. Another stretch goal was to use a colored image rather than a gray scale one. After constructing this project with gray scale and observing high memory usage, there would have be optimizations in other areas that would allow for colored images to run through multiple filters properly.

VII. Retrospective

- The user interface was an important part of the project's experience. To best design it, there was careful consideration in place to map out all coordinates such that the images displayed nicely. Throughout the process, a lot of these values changed due to adding new constraints such as more image sprites to the screen. When these values were changed, it became difficult to keep track of where every instance was, and it was also a struggle to ensure that my group mates' code had the same values. In the future, there should have been parameters established early on to hard code those constants.
- Due to time constraints, there was average effort made to pipeline the system entirely. Currently, the design still contains some artifacting and stray lines that should not be present. As the system design evolved, ensuring that timing was met became more difficult.
- Originally, it was outlined that there would be image manipulation within the project, and there indeed is. One of the filters creates a wave effect on an image. It would have been nice to have included more image manipulation, such as a spiral or kaleidoscope effect. With more time, this would have been a path to consider since it would have provided the project with more advanced functionality.
- For the sake of the project, the camera was sufficient, but if this was to be extended to other users, a higher quality camera would be required. With a higher quality camera, filters would have appeared nicer and thresholding would have provided much more drastic differences.
- Within the code base itself, there could have been better coding conventions. For example, there were a few modules and files that were not utilized and could have been deleted to clean up the folder. Additionally, some modules could have interfaced with others. There was also some duplicated code within modules that could have been reduced by using better modularity and parametrization.
- Above all, starting to integrate much earlier would have been the best thing for the group. As time ran out and things became rushed, it was difficult to combine everything while having to wait a long time for builds and to debug as well.

VIII. Acknowledgements

For this project, Kojo focused mainly on the user interface, Diego was tasked with image processing, and Ellie worked with the hardware and plotting modules. Kojo and Diego paired up to ensure that the filters could lay atop of the user interface. We would like to thank Joe for being an engaging professor, Jay and Fischer for being great TAs, and the other LAs who have dedicated plenty of their time to help with labs, psets, and projects.

IX. References

https://github.com/rabenold/6.111FinalProject